Implementation of a Web Development Platform for Embedded System Designers

Milan Raj
IoT Software Technologies
National Instruments
Austin, Texas, United States
milan.raj@ni.com

***Abstract*—Distributed intelligent embedded devices rely on continuous feature improvements and on-demand user interactions to meet the evolving requirements of distributed systems. Technologies like HTML, CSS, and JavaScript are quickly becoming the de facto standard for developing cross-platform software. However, most embedded systems developers, despite proficiency in low-level programming, find web development unapproachable due to unfamiliar programming patterns, standards, frameworks and tools. The use of proprietary technologies or constrained platforms limits the ability of embedded developers to keep up with changing requirements in distributed applications. This paper explores the implementation of a next generation, open HMI development platform based on web standards and open source codebases.**

Keywords—HTML; JavaScript; CSS; WYSIWYG; asm.js; HTTP; HMI; Web

I. INTRODUCTION

Despite the introduction of many new user interface technologies and application development practices over the years, the web browser has maintained consistent availability across practically every device with rich graphical user interfaces. Active efforts by browser developers to improve interoperability [1] and focus on development of low-level primitives [2] have shifted the role of the browser from an interactive document scripting environment to a high-performance application virtual machine.

We demonstrate how to leverage open standards and community driven developments to minimize risk of platform divergence and build an adaptable code base capable of adopting new standards as they stabilize. We present a WYSIWYG web-based HMI development platform for embedded system designers and discuss our strategies for web technology selection.

II. SYSTEM ARCHITECTURE OVERVIEW

A development environment hosting a modern embedded web browser is used to enable the WYSIWYG creation of HTML user interfaces. A user of the development environment can write logic for the corresponding HTML UI using a graphical dataflow programming language to read from and write values to HTML UI elements. The user is also capable of using the graphical programming language to perform HTTP network requests and perform parsing and analysis of plain text and JSON responses. The HTML UI and dataflow programming diagram are both contained in a source file format operating as a functional unit and referred to as a Web Virtual Instrument. Multiple Web Virtual Instrument files can reference each other and are compiled together to generate a complete web application consisting of an application specific HTML file and a Virtual Instrument Assembly file containing a text representation of the transformed and merged dataflow programming diagrams. In addition to application specific files, static resources are generated containing JavaScript, CSS, and other media assets that are consistent between each generated web application.

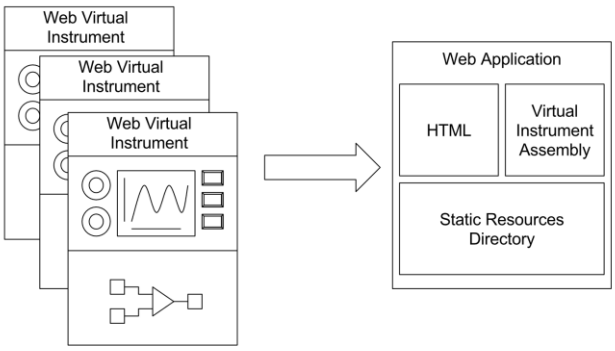

Fig. 1. Compilation of multiple Web Virtual Instrument source files to a standalone web application.

As an open platform, the development environment creates web applications capable of communicating with arbitrary HTTP-based web services. Included with the development environment is a data services platform to facilitate communication between generated web applications and embedded devices. The data services platform consists of HTTP endpoints queryable from generated web applications and allows for reading and writing named variables with latest values and subscribing to named queues of messages. An embedded device on a network can use HTTP or AMQP to communicate with the data services platform to publish the current device state or subscribe to message queues.

A complete HMI solution may consist of a web application generated by the development environment that communicates with the data services platform for monitoring and control of network connected embedded systems. An operator can use commodity devices with modern web browsers to access the generated web applications. The focus of this paper will be on the architecture of the generated web applications which utilize open standards to implement modern user interfaces and implement client-side logic driven by a high-level graphical programming language.

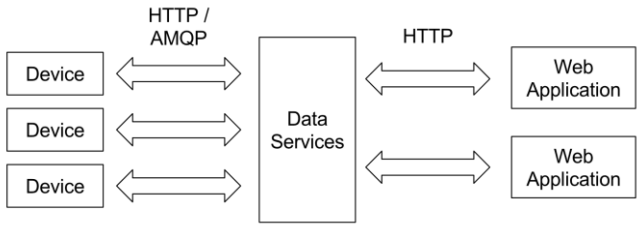

Fig. 2. Example configuration for a complete HMI solution.

III. GENERATED WEB APPLICATION ARCHITECTURE

A. Overview of Deployed Files

Each built web application consists of an HTML file with configuration used for the user interface of the application and a Virtual Instrument Assembly text file containing a low-level text representation of the user-developed dataflow programming diagram. In addition, there is a static resources directory containing the JavaScript files for implementing the HTML UI controls, CSS for HTML UI control theming, JavaScript implementing a dataflow programming language runtime, and other resources used such as images and localization files.

The primary entry point of the web application is the HTML file. The HTML file contains references to the resources previously described, an inline stylesheet corresponding to the HTML UI controls the user interactively placed in the WYSIWYG editor, and a series of custom HTML elements with attributes that represent the configured state of the HTML UI controls.

B. Custom Element Utilization

Custom Elements are a technology that allow user-defined custom HTML tags to be registered in a web application. These custom HTML tags enable new types of user interface elements to be added to a web page by inserting the newly defined tag in the HTML. The author of a Custom Element can observe when the element is inserted or removed from the HTML or when the element's HTML attributes are modified [3]. Custom Elements behave like other HTML Elements in the Document Object Model (DOM) of a web page. Like other HTML Elements natively supported by a web browser, each Custom Element has HTML attributes, JavaScript properties and methods, and the ability to fire and listen for events triggered in the DOM hierarchy.

For web applications generated by the development environment, the entire state of each HTML UI control is captured in the respective HTML attributes of their Custom Element. The benefit of this approach over having HTML UI control configuration stored in generated JavaScript or other data formats is improved embeddability and styling customizability. The Custom Elements can be moved around, styled, and manipulated like any other HTML element and hold their configuration information. These properties also make Custom Element-based HTML UI controls highly reusable in other web applications.

C. Modeling and Framework

During the startup of a generated web application the lifecycle events of Custom Elements are monitored by a JavaScript framework that implements semantics like the Model-View-ViewModel (MVVM) pattern. After a Custom Element is connected to the DOM the framework triggers the creation of a Model object and ViewModel object that corresponds to the View represented by the Custom Element.

In the MVVM framework, the Model provides an abstraction over the representation of the properties for a HTML UI control. The Model representation of a control's properties provide a consistent interface for other agents such as the desktop environment or embedded dataflow runtime to send updates. When an update occurs to a Model, the corresponding ViewModel updates a Render object representing a mutation action for the corresponding HTML UI control. The Render objects are queued into the Render engine that is serviced on a requestAnimationFrame [4] callback for the web page. The net effect is that high frequency Model updates can be performed which are collated and serviced at optimal rendering times across all Custom Elements managed by the MVVM framework.

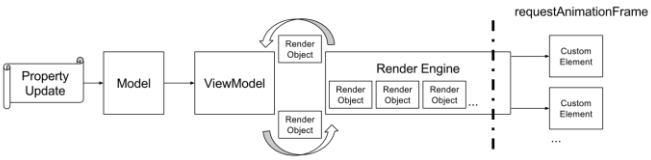

Fig. 3. Flow of a property update through the MVVM framework to apply to a Custom Element.

D. Application Update Service Management

In addition to having Custom Elements to capture the state and initialize modeling for HTML UI controls, there are also nonvisible Custom Elements used to store the configuration for update service management. The update services are state machines tasked with transitioning the state of a web application from page load through completion. Different update services are implemented for the different environments and expected behaviors of the web application.

Two of the most significant update services are the editor update service and the local update service:

The editor update service is utilized when the web application is running inside the development environment and is expected to respond to WYSIWYG editing operations. As the user performs editing operations such as changing size, position, or configuration of a control, messages are sent asynchronously from the development environment to the embedded web browser hosting the web application. The editor

update service receives the messages and applies the updates to the Models targeted by the development environment.

The local update service is used when a user tests the execution of their web application in the development environment or when the web application is deployed and running in a standalone web browser. In this configuration the local update service has the responsibility of fetching the application-specific Virtual Instrument Assembly file, passing the Virtual Instrument Assembly file contents to the bundled dataflow runtime environment, and mediating control updates between the dataflow runtime environment and the Models.

E. Virtual Instrument Runtime Engine Object (Vireo)

Vireo is an open source dataflow runtime used in the web application to execute the instructions stored in Virtual Instrument Assembly files. The runtime is a compact C++ project capable of managing memory and scheduling execution of the low-level dataflow programs created by the user in the development environment. The runtime has been designed for execution in resource constrained embedded systems giving it a small size and memory footprint suitable for web applications.

To make Vireo executable in the browser environment we leveraged the open source Emscripten toolchain [5] to compile C++ source to a subset of the JavaScript language known as asm.js [6]. The asm.js subset of JavaScript makes an efficient target for compilers by restricting code to use primarily math operations and to perform those operations on one large shared JavaScript ArrayBuffer. Benchmarking has shown that C++ projects compiled to asm.js and running on modern JavaScript browser runtimes execute within two-thirds the speed, or better, of the same C++ projects compiled using Clang or GCC and executing natively [7].

IV. PERFORMANCE CHARACTERISTICS OF WEB-BASED CONTROLS

A common concern based on historical JavaScript execution behavior and conflated with poor usage patterns of the browser DOM API is that HTML UI controls may be unable to maintain fast and responsive user interfaces. What we have observed is that we can demonstrate desirable performance characteristics when we approach development of HTML UI elements with the same rigor we would use in other user interface environments.

The best demonstration of performance characteristics for HTML UI controls comes from the graphing and charting Custom Element implementations. These Custom Elements were implemented by leveraging the existing open source Flot charting library [8] and creating an open source fork, known as engineering-flot, with features and optimizations well-suited for engineering and scientific applications [9].

In the Custom Elements built using the engineering-flot codebase we utilize the HTML5 canvas element for drawing, prevent unnecessary copies of buffers, and implement data decimation algorithms to avoid unnecessary drawing operations. Benchmarking of the graph Custom Elements on modern desktop web browsers has shown the capability of rendering over 500,000 data points per frame at sixty frames per second [10].

V. CONSIDERATIONS FOR ADOPTION OF NEW WEB TECHNOLOGIES

Selecting features of the modern web platform to adopt in a new project has additional considerations compared to traditional desktop application development or even traditional web development. Historically web browser versioning was highly coupled to the operating system platform and operating system version for which the browsers were designed. This coupling made it common to use browser version of the target audience as the primary consideration for choosing which features to adopt during web application development. Many modern browsers release frequently and can have releases performed independently from their hosted operating system. These continuously updated browsers are referred to as "evergreen" browsers and can result in benefits such as improved interoperability with other browsers [11].

With browsers updating frequently, decoupled from the underlying operating system, and containing increasingly interoperable sets of shared features, it becomes possible to change the web platform feature selection process for new application development. Instead of choosing a browser and opting into all the web platform features that browser implements, it is possible to choose a feature and see if it is implemented in all the browsers you choose to support.

If there is a subset of browsers that do not support a feature it may be possible to utilize a polyfill for the feature. A polyfill is code that attempts to implement a feature that might be missing from a browser where existing browser features can be used to replicate or closely approximate the missing feature [12]. From our experience, a well-designed and low-risk polyfill may have some, or all, of the following characteristics:

- Compact in code size in a shipping web application
- Comparable performance to the feature as natively implemented in a browser
- Closely implements the native browser feature with few polyfill-specific exceptions
- Makes well-understood changes to the browser global environment
- Represents a specification that browsers follow or have committed to follow in the future
- Delegates execution of the feature to the native implementation if available
- Removable with little to no changes in source code as browsers enable the feature natively

As opposed to consuming libraries that try to abstract over differences between browsers by providing a nonstandard API on top of those differences, polyfills attempt to bring up the baseline usable set of features across browsers. While custom libraries lead to an increasingly siloed ecosystem of libraries interdependent on nonstandard APIs, utilizing polyfills that are backed by open standards leads to the polyfills becoming removable as they are backed by native browser implementations over time.

VI. SUMMARY

We have discussed an architecture for enabling development of web applications supporting WYSIWYG manipulation using an MVVM style framework to schedule HTML UI control updates in a performant manner. It is described how Custom Elements are used to hold UI configuration and form the basis of reusable HTML UI controls that are highly customizable by users in a deployed application. In addition, nonvisible Custom Elements also maintain configuration in HTML attributes creating a consistent interface for all configuration of the web application.

We demonstrated the ability to use open source tooling to leverage existing C++ code through compilation to the asm.js JavaScript subset and described results of highly performant and responsive HTML UI graph controls. Finally, we presented an approach for selecting web platform features for use in new web application development by selecting features backed by open standards with native browser implementations or with implementations that can be backed by well-designed polyfills.

REFERENCES

[1] *About*, Mozilla, webcompat.com/about.

[2] *The Extensible Web Manifesto*, Extensibleweb, extensiblewebmanifesto.org/.

[3] Denicola, Domenic. "Custom Elements". W3C, 13 Oct. 2016, w3.org/TR/custom-elements/#custom-element-reactions.

[4] "Window.requestAnimationFrame()." *Mozilla Developer Network*, Mozilla, 28 Nov. 2017, developer.mozilla.org/en-US/docs/Web/API/window/requestAnimationFrame.

[5] "Emscripten." *GitHub*, github.com/kripken/emscripten.

[6] Herman, David, Zakai, Alon, and Luke Wagner. *Asm.js*, Mozilla, 18 Aug. 2014, asmjs.org/spec/latest/.

[7] Zakai, Alon, and Robert Nyman. "Gap between asm.Js and native performance gets even narrower with float32 optimizations – Mozilla Hacks - the Web developer blog." *Mozilla Hacks – the Web developer blog*, Mozilla, 20 Dec. 2013, hacks.mozilla.org/2013/12/gap-between-asm-js-and-native-performance-gets-even-narrower-with-float32-optimizations/.

[8] "Flot." *GitHub*, github.com/flot/flot.

[9] "Engineering-flot." *GitHub*, github.com/ni-kismet/engineering-flot

[10] "Creating Web Enabled HMIs with LabVIEW NXG." Performance by Mark Black, Eli Kerry, and Omid Sojoodi, *Creating Web Enabled HMIs with LabVIEW NXG*, National Instruments, 23 May 2017, youtube.com/watch?v=N4XCNfGapc4&t=1m17s.

[11] Beeman, Hadley. "The evergreen Web." W3C, 9 Feb. 2017, w3.org/2001/tag/doc/evergreen-web/.

[12] Lawson, Bruce, and Remy Sharp. *Introducing HTML5*. New Riders, 2012, pp. 276-277.